1 Honor Code

Declare and sign the following statement:

"I certify that all solutions in this document are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

Signature : Cheng-Han Chuang

While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that consequences of academic misconduct are *particularly severe*!

Students : None

2.

$$\max_{\lambda_i \geq 0} \min_{w,\alpha} \; \|w\|^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1). \tag{3}$$

Note: λ_i must be greater than or equal to 0.

(a) Show that equation (3) can be rewritten as the *dual optimization problem*

$$\max_{\lambda_i \geq 0} \sum_{i=1}^{n} \lambda_i - \frac{1}{4} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j X_i \cdot X_j \text{ subject to } \sum_{i=1}^{n} \lambda_i y_i = 0. \tag{4}$$

Hint: Use calculus to determine and prove what values of w and α optimize equation (3). Explain where the new constraint comes from.

(b) Suppose we know the values λ_i^* and α^* that optimize equation (3). Show that the decision rule specified by equation (1) can be written

$$r(x) = \begin{cases} +1 & \text{if } \alpha^* + \frac{1}{2} \sum_{i=1}^{n} \lambda_i^* y_i X_i \cdot x \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{5}$$

(c) Applying Karush–Kuhn–Tucker (KKT) conditions (See Wikipedia for more information), any pair of optimal primal and dual solutions w^*, α^*, λ^* for a linear, hard-margin SVM must satisfy the following condition:

$$\lambda_i^* (y_i (X_i \cdot w^* + \alpha^*) - 1) = 0 \;\; \forall i \in \{1, \dots, n\}$$

This condition is called *complementary slackness*. Explain what this implies for points corresponding to $\lambda_i^* > 0$.

(d) The training points X_i for which $\lambda_i^* > 0$ are called the *support vectors*. In practice, we frequently encounter training data sets for which the support vectors are a small minority of the training points, especially when the number of training points is much larger than the number of features. Explain why the support vectors are the only training points needed to evaluate the decision rule.

(e) The obtained parameters when fitting the linear SVM to the 2D synthetic dataset found in **toy-data.npz** approximately correspond to

$$w = \begin{bmatrix} -0.4528 \\ -0.5190 \end{bmatrix} \quad \text{and} \quad \alpha = 0.1471. \tag{6}$$

Using only matplotlib basic plotting functions, in your write-up, produce a plot of

- the data points,
- the decision boundary,
- the margins, defined as $\{x \in \mathbb{R}^2 : w \cdot x + \alpha = \pm 1\}$.

In this plot, where are the support vectors?

Hint: You can use the following snippet, that plots the data points and decision boundary but not the margins.

```
plt.scatter(data[:, 0], data[:, 1], c=labels)

# Plot the decision boundary
x = np.linspace(-5, 5, 100)
y = -(w[0] * x + b) / w[1]
plt.plot(x, y, 'k')

# Plot the margins
## TODO
```

$\|w'\| \leq \|w\|$

(f) Assume the training points X_i and labels y_i are linearly separable. Using the original SVM formulation (not the dual) prove that there is at least one support vector for each class, +1 and −1.

Hint: Use contradiction. Construct a new weight vector $w' = w/(1 + \epsilon/2)$ and corresponding bias α' where $\epsilon > 0$. It is up to you to determine what ϵ should be based on the contradiction. If you provide a symmetric argument, you need only provide a proof for one of the two classes.

(a) Take the gradient of (3) wrt W, we get $2W - \sum_{i=1}^{n} \lambda_i y_i x_i$, setting this to 0, our $W^* = \frac{1}{2}\sum_{i=1}^{n} \lambda_i y_i x_i$

Taking the second gradient gives us 2, which is positive, meaning the equation is convex, thus the critical point is a minimum point

Take the gradient of (3) wrt α and also setting it to 0, gives us $0 = \sum_{i=1}^{n} \lambda_i y_i$

Replacing w^* in (3) gives us

$$\max_{\lambda_i \geq 0} \frac{1}{4} \sum_{i}^{n} \sum_{i}^{n} \lambda_i \lambda_i y_i y_i X_i \cdot X_i - \sum \lambda_i y_i X_i \left(\frac{1}{2} \sum \lambda_i y_i X_i \right) + \sum_{i=1}^{n} \lambda_i$$

$$= \max_{\lambda_i \geq 0} \sum_{i}^{n} \lambda_i - \frac{1}{4} \sum_{i}^{n} \sum_{i}^{n} \lambda_i \lambda_i y_i y_i X_i \cdot X_i$$ with the constraint $\sum_{i=1}^{n} \lambda_i y_i = 0$, which is (4)

(b) Do this by substituting w^* into (1), which gives $r(x) = \begin{cases} +1 & \text{if } \alpha^* + \underbrace{\frac{1}{2}\sum_{i=1}^{n} \lambda_i y_i X_i}_{\to\, w^*} \cdot x \geq 0 \\ -1 & \text{o/w} \end{cases}$

(c) Since by equation (2), $y_i(X_i \cdot w + \alpha) \geq 1$, if $\lambda^* > 0$, that means $y_i(X_i \cdot w^* + \alpha^*) - 1 = 0$, which by definition means the point is on the margin. So when $\lambda_i^* > 0$, the points are on the margin.

(d) Non support vector points have $\lambda_i^* = 0$, so only α determines the decision rule in part (b), thus only support vectors contribute to the decision rule

(f) By contradiction, WLOG assume there is no support vector in class +1, then there's no point on the boundary, so $X_i \cdot w + \alpha > 1$, set ϵ to be the smallest distance of these points to margin, or $\epsilon = \min X_i \cdot w + \alpha - 1$

Let's prove this wrong by finding a w' and α' with a lower cost: larger margin, smaller weight
min. overfitting

As stated in the hint, let $w' = w/(1+\epsilon/2)$

With a support vector, point X_i would have constraint $X_j \cdot w' + \alpha' = 1$.

Given $\begin{cases} X_j \cdot w' + \alpha' = 1 \\ X_j \cdot w + \alpha = 1 + \epsilon \end{cases}$ $\quad \alpha' = 1 - (X_j \cdot w)/(1 + \epsilon/2) = \dfrac{\alpha - \epsilon/2}{1 + \epsilon/2}$

with $\|w'\| < \|w\|$ we've found a better classifier

hw1_write

January 24, 2024

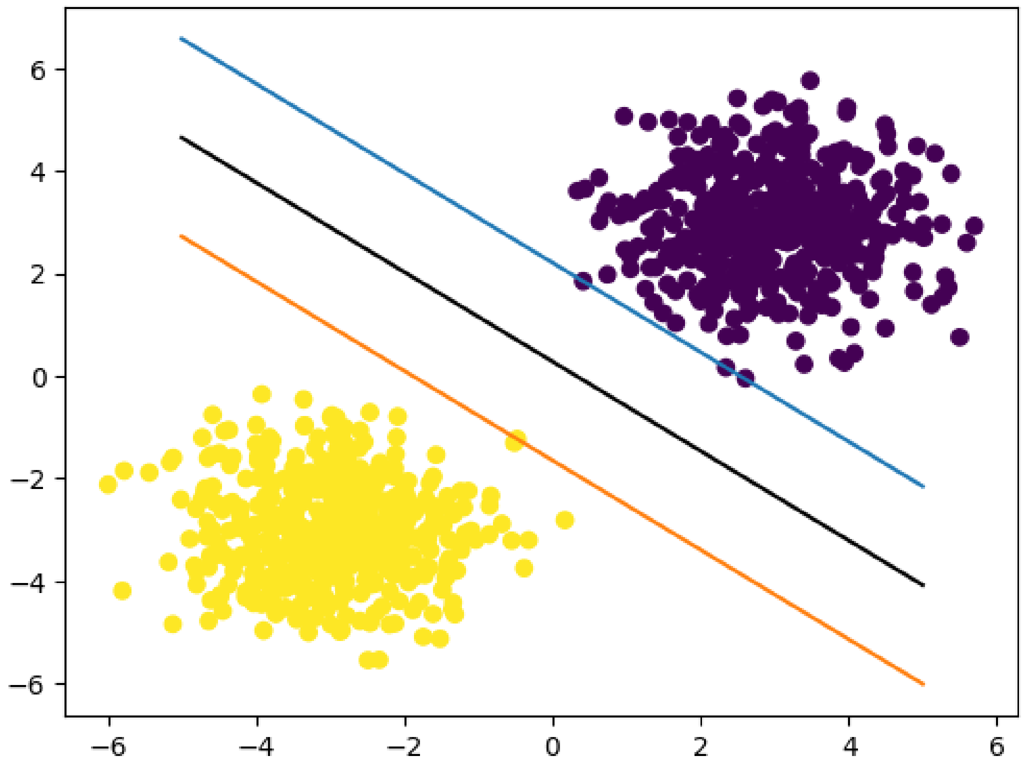
[ ]:
```
import sys
if sys.version_info[0] < 3:
   raise Exception("Python 3 not detected.")
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from scipy import io
if __name__ == "__main__":
    for data_name in ["mnist", "spam", "toy"]:
       data = np.load(f"data/{data_name}-data.npz")
       print("\nloaded %s data!" % data_name)
       fields = "test_data", "training_data", "training_labels"
       for field in fields:
          print(field, data[field].shape)
```

```
loaded mnist data!
test_data (10000, 1, 28, 28)
training_data (60000, 1, 28, 28)
training_labels (60000,)

loaded spam data!
test_data (1000, 43)
training_data (4171, 43)
training_labels (4171,)

loaded toy data!
test_data (0,)
training_data (1000, 2)
training_labels (1000,)
```

0.1 2.e Toys

[ ]:
```
toy = np.load("data/toy-data.npz")

toy_train_data = toy["training_data"]
toy_train_labels = toy["training_labels"]
w = [-0.4528, -0.5190]
```

```
alpha = 0.1471

# Plot the points
plt.scatter(toy_train_data[:, 0], toy_train_data[:, 1], c=toy_train_labels)

# Plot decision boundary
x = np.linspace(-5, 5, 100)
y = -(w[0] * x + alpha) / w[1]  # rearranged from w[0]*x1 + w[1]*x2 + alpha
plt.plot(x, y, "k")

# Plot margins
y_margin_lower = -((w[0] * x + alpha) - 1) / w[1]
y_margin_upper = -((w[0] * x + alpha) + 1) / w[1]
plt.plot(x, y_margin_upper)
plt.plot(x, y_margin_lower)
plt.show()
```



0.2 3. Data Partition

```
[ ]: np.random.seed(15)

     """Shuffles and partitions data"""


     def partition(data, labels, validation_size):
         total_size = len(data)
         # in the case where a percentage is given
         if validation_size < 1:
             validation_size = int(validation_size * total_size)
         shuffled_ind = np.random.permutation(total_size)

         # uses fancy indexing, first reshuffling data, then getting the validation␣
      ↪set
         val_data = data[shuffled_ind][:validation_size]
         val_label = labels[shuffled_ind][:validation_size]
         train_data = data[shuffled_ind][validation_size:]
         train_label = labels[shuffled_ind][validation_size:]

         return train_data, train_label, val_data, val_label


     def eval_metric(y_pred, y_hat):
         assert len(y_pred) == len(y_hat)
         total_pred = len(y_pred)
         correct_pred = np.sum(y_pred == y_hat)  # vectorized, easier than for loop
         return correct_pred / total_pred
```

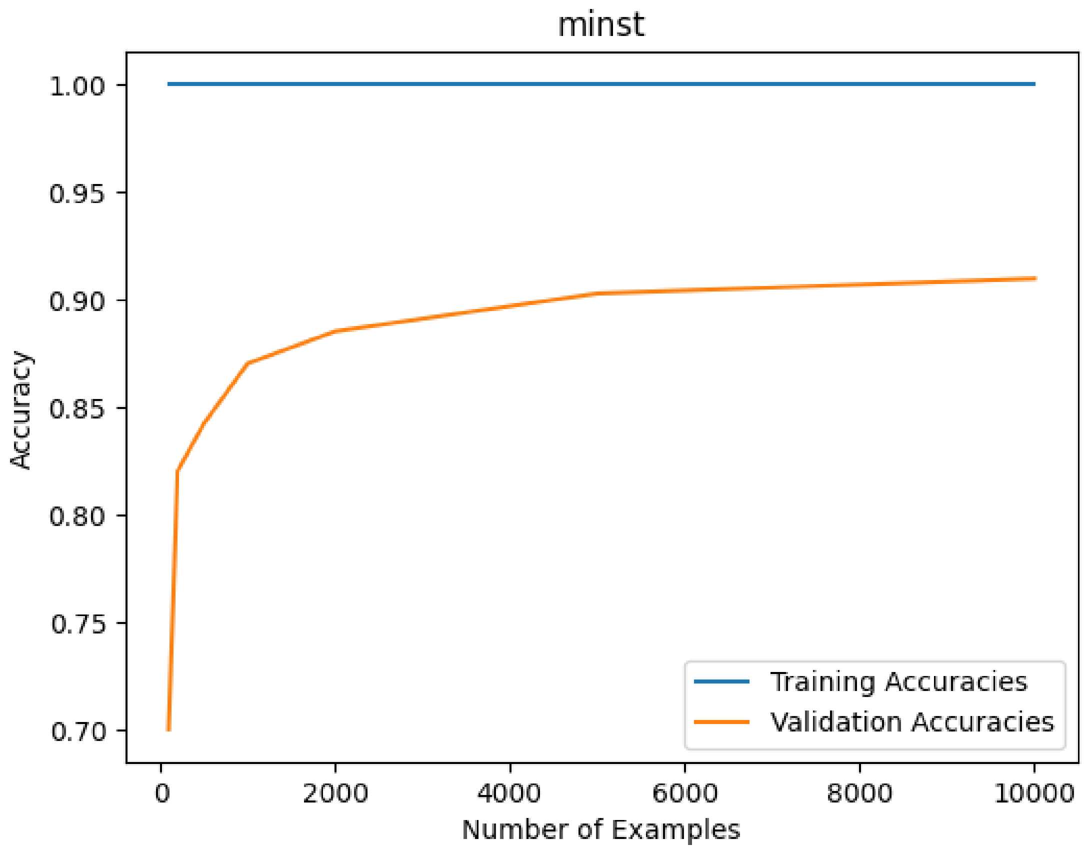
```
[ ]: mnist = np.load(f"data/mnist-data.npz")
     spam = np.load(f"data/spam-data.npz")

     mnist_train_data_flat = mnist["training_data"].reshape(
         mnist["training_data"].shape[0], -1
     )

     minst_train_d, minst_train_l, minst_val_d, minst_val_l = partition(
         mnist_train_data_flat, mnist["training_labels"], 10000
     )
     spam_train_d, spam_train_l, spam_val_d, spam_val_l = partition(
         spam["training_data"], spam["training_labels"], 0.2
     )

     print(
         len(minst_train_l),
         len(minst_val_l),
    len(minst_val_d),
    len(spam_train_l),
    len(spam_train_d),
    len(spam_val_l),
)
```

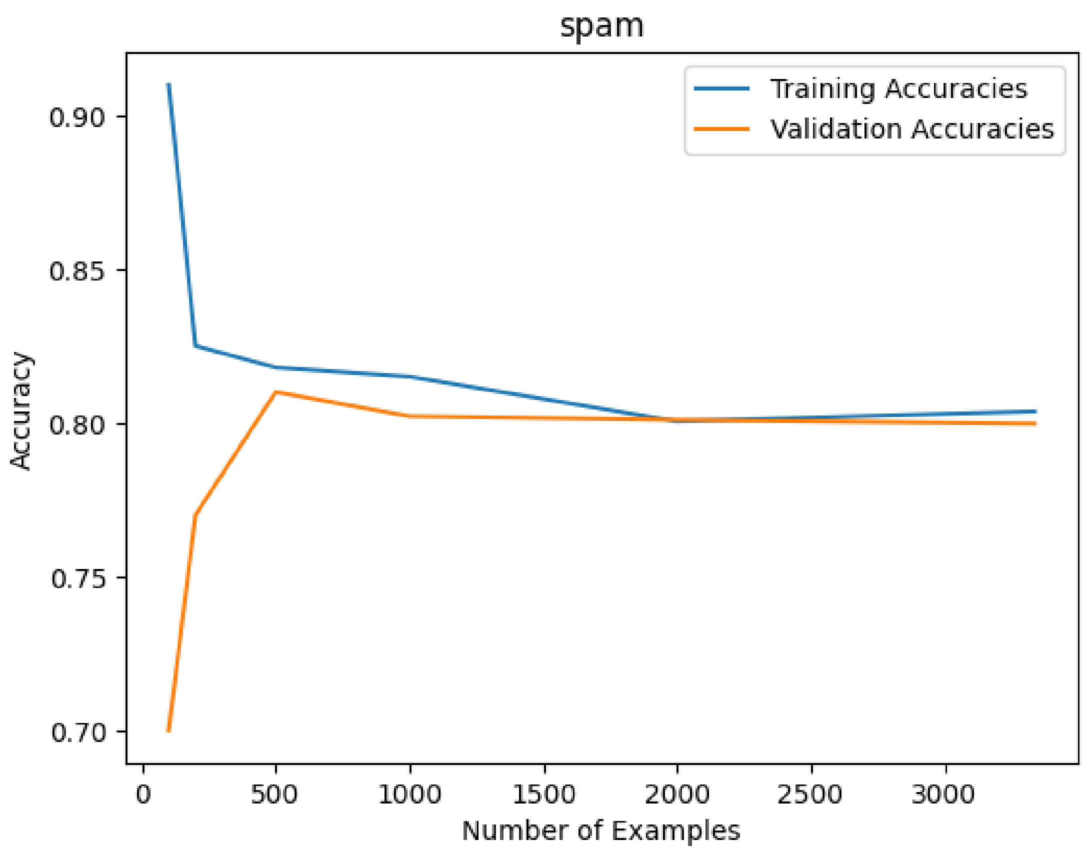
```
50000 10000 10000 3337 3337 834
```

0.3 4. Support Vector Machines

[ ]:
```
def svm_model(data, training_sizes, train_data, train_label, val_data,␣
 ↪val_label):
    # store accuracies across diff training sizes
    train_accuracies = []
    val_accuracies = []
    for size in training_sizes:
        # Train model
        model = svm.SVC(kernel="linear")
        model.fit(train_data[:size], train_label[:size])

        # Predict training accuracy
        train_pred = model.predict(train_data[:size])
        train_accuracy = eval_metric(train_pred, train_label[:size])
        train_accuracies.append(train_accuracy)

        # Predict validation accuracy
        val_pred = model.predict(val_data[:size])
        val_accuracy = eval_metric(val_pred, val_label[:size])
        val_accuracies.append(val_accuracy)
    plot_ex_accuracies(data, training_sizes, train_accuracies, val_accuracies)


def plot_ex_accuracies(data, training_sizes, train_accuracies, val_accuracies):
    plt.figure()
    plt.plot(training_sizes, train_accuracies, label="Training Accuracies")
    plt.plot(training_sizes, val_accuracies, label="Validation Accuracies")
    plt.title(data)
    plt.xlabel("Number of Examples")
    plt.ylabel("Accuracy")
    plt.legend()
```

[ ]:
```
m_train_sizes = [100, 200, 500, 1000, 2000, 5000, 10000]
s_train_sizes = [100, 200, 500, 1000, 2000, spam_train_d.shape[0]]

# print("loaded mnist data!")
# fields = "test_data", "training_data", "training_labels"
# for field in fields:
#     print(field, mnist[field].shape)

# print(mnist_train_data_flat.shape)



svm_model(
    "minst", m_train_sizes, minst_train_d, minst_train_l, minst_val_d,␣
 ↪minst_val_l
)
svm_model("spam", s_train_sizes, spam_train_d, spam_train_l, spam_val_d,␣
 ↪spam_val_l)
```




spam
Training Accuracies
Validation Accuracies
Accuracy
0.70
0.75
0.80
0.85
0.90
0
500
1000
1500
2000
2500
3000
Number of Examples

0.4 5. Hyperparameter Tuning

I tried these C values [0.00000001, 0.0000001, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.05, 0.1, 1], the corresponding accuracy was 1e-08 : 0.8883 1e-07 : 0.9222 1e-06 : 0.9298 1e-05 : 0.918 0.0001 : 0.9095 0.001 : 0.9095 0.01 : 0.9095 0.05 : 0.9095 0.1 : 0.9095 1 : 0.9095 The optimal value for C is 1e-06, with an acurracy of 0.9298

[ ]:
```python
# Testing different C values
def hyper_training(train_size, train_data, train_label, val_data, val_label,␣
↪c_values):
    val_accuracies = []

    for c in c_values:
        # Train model
        hyper_model = svm.SVC(kernel="linear", C=c)
        hyper_model.fit(train_data[:train_size], train_label[:train_size])

        # Predict validation accuracy
        val_pred = hyper_model.predict(val_data)
        val_accuracy = eval_metric(val_pred, val_label)
        val_accuracies.append((c, val_accuracy))
        print(f"C={c}: Cross-validation accuracy = {val_accuracy}")
    # print(val_accuracies)
```

[ ]:
```python
hyper_training(
    10000,
    minst_train_d,
    minst_train_l,
    minst_val_d,
    minst_val_l,
    [0.00000001, 0.0000001, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.05, 0.1,␣
↪1]
)
```

```
1e-08 :  0.8883
1e-07 :  0.9222
1e-06 :  0.9298
1e-05 :  0.918
0.0001 :  0.9095
0.001 :  0.9095
0.01 :  0.9095
0.05 :  0.9095
0.1 :  0.9095
1 :  0.9095
```

0.5 6. K-fold cross validation

I tried the C values listbed below and these were the accuracies, the best C value is 10 in my case C=1e-05: Cross-validation accuracy = 0.7123028762618647 C=0.0001: Cross-validation accuracy = 0.7168577952009649 C=0.001: Cross-validation accuracy = 0.7487428021654533 C=0.01: Cross-validation accuracy = 0.7753540401211965 C=0.05: Cross-validation accuracy = 0.791898217952584 C=0.1: Cross-validation accuracy = 0.7926170680222289 C=1: Cross-validation accuracy = 0.8005293011100102 C=10: Cross-validation accuracy = 0.8036456583236407 C=50: Cross-validation accuracy = 0.8029265210586022 C=100: Cross-validation accuracy = 0.8034061373655568

```
[ ]: def k_fold_cross_validation(train_data, train_label, c_values, k=5):
         total_size = len(train_data)
         shuffled_ind = np.random.permutation(total_size)

         fold_size = total_size // k
         C_accuracies = []

         for c in c_values:
             fold_accuracies = []
             for fold in range(k):
                 # indices for val set
                 start, end = (
                     fold * fold_size,
                     (fold + 1) * fold_size if fold < k - 1 else total_size,
                 )

                 # val, training indices
                 val_indices = shuffled_ind[start:end]
                 train_indices = np.concatenate((shuffled_ind[:start],␣
      ↪shuffled_ind[end:]))

                 # slice data
                 train_fold_data, train_fold_label = (
                     train_data[train_indices],
                     train_label[train_indices],
                 )
                 val_fold_data, val_fold_label = (
                     train_data[val_indices],
                     train_label[val_indices],
                 )

                 k_fold_model = svm.SVC(kernel="linear", C=c)
                 k_fold_model.fit(train_fold_data, train_fold_label)
                 val_predict = k_fold_model.predict(val_fold_data)
                 val_accuracy = eval_metric(val_predict, val_fold_label)
                 fold_accuracies.append(val_accuracy)
             avg_accuracy = np.mean(fold_accuracies)
```

```
        C_accuracies.append((c, avg_accuracy))
        print(f"C={c}: Cross-validation accuracy = {avg_accuracy}")

    C_accuracies.sort(
        key=lambda x: x[1], reverse=True
    )  # sort by descending avg_accuracy
    return C_accuracies
```

[ ]:
```
spam = np.load(f"data/spam-data.npz")

k_fold_cross_validation(
    spam["training_data"],
    spam["training_labels"],
    [0.00001, 0.0001, 0.001, 0.01, 0.05, 0.1, 1, 10, 50, 100],
)
```

```
C=1e-05: Cross-validation accuracy = 0.7123028762618647
C=0.0001: Cross-validation accuracy = 0.7168577952009649
C=0.001: Cross-validation accuracy = 0.7487428021654533
C=0.01: Cross-validation accuracy = 0.7753540401211965
C=0.05: Cross-validation accuracy = 0.791898217952584
C=0.1: Cross-validation accuracy = 0.7926170680222289
C=1: Cross-validation accuracy = 0.8005293011100102
C=10: Cross-validation accuracy = 0.8036456583236407
C=50: Cross-validation accuracy = 0.8029265210586022
C=100: Cross-validation accuracy = 0.8034061373655568
C=1000: Cross-validation accuracy = 0.8036459455190339
```

[ ]:
```
[(1000, 0.8036459455190339),
 (10, 0.8036456583236407),
 (100, 0.8034061373655568),
 (50, 0.8029265210586022),
 (1, 0.8005293011100102),
 (0.1, 0.7926170680222289),
 (0.05, 0.791898217952584),
 (0.01, 0.7753540401211965),
 (0.001, 0.7487428021654533),
 (0.0001, 0.7168577952009649),
 (1e-05, 0.7123028762618647)]
```

0.6 Question 7: Kaggle Submissions

My submission on Kaggle had a score of Mnist: 0.978, and Spam: 0.84. For both datasets, I wrote a function that finds the best model and hyperparameters for me, experiementing on different kernels: poly, rbf; different values of C, different values of gamma and so on. By sorting though this I was able to find models for mnist that worked really well using “poly”, C = 0.000001, gamma = 0.01. However, there doesn’t seem to be too much of an improvement in Spam, especially since k-fold modeling wasn’t available (can only submit 1 model). So for Spam, I read through some of the emails and experimented with adding features such as “credit”, “$”, “offer”, and used my general intuition on spam/ham emails to add even more features like “click”, “urgent”. By adding suich features and tuning hyperparameters and using rbf, I was able to achieve a 84% score for spam.

```
[ ]: def find_best_model(dataset, train_data, train_label, val_data, val_label, c):
         val_accuracies = []
         gammas = [0.001, 0.01, 0.1]
         kernels = ["poly", "rbf"]
         for kernel in kernels:
             for gamma in gammas:
                 val_accuracy = train_model(
                     dataset,
                     train_data,
                     train_label,
                     val_data,
                     val_label,
                     kernel,
                     c,
                     gamma,
                 )
                 val_accuracies.append((val_accuracy, kernel, c, gamma))
                 print(f"Accuracy: {val_accuracy}, kernel: {kernel}, C={c}, gamma:␣
      ↪{gamma}")
         return val_accuracies.sort(key=lambda x: x[1], reverse=True)


     def train_model(
         dataset, train_data, train_label, val_data, val_label, kernel, c, gamma
     ):
         # if dataset == "spam":
         #     accuracy = k_fold_cross_validation(kernel, c, gamma, train_data,␣
      ↪train_label)
         #     return accuracy
         # else:
             model = svm.SVC(kernel=kernel, C=c, gamma=gamma)
             model = model.fit(train_data, train_label)
             val_predict = model.predict(val_data)
             val_accuracy = eval_metric(val_predict, val_label)
             return val_accuracy
```

```
# def k_fold_cross_validation(kernel, c, gamma, train_data, train_label, k=5):
#     total_size = len(train_data)
#     shuffled_ind = np.random.permutation(total_size)

#     fold_size = total_size // k

#     fold_accuracies = []
#     for fold in range(k):
#         # indices for val set
#         start, end = (
#             fold * fold_size,
#             (fold + 1) * fold_size if fold < k - 1 else total_size,
#         )

#         # val, training indices
#         val_indices = shuffled_ind[start:end]
#         train_indices = np.concatenate((shuffled_ind[:start],␣
 ↪shuffled_ind[end:]))

#         # slice data
#         train_fold_data, train_fold_label = (
#             train_data[train_indices],
#             train_label[train_indices],
#         )
#         val_fold_data, val_fold_label = (
#             train_data[val_indices],
#             train_label[val_indices],
#         )

#         k_fold_model = svm.SVC(kernel=kernel, C=c, gamma=gamma)
#         k_fold_model.fit(train_data, train_label)
#         val_predict = k_fold_model.predict(val_fold_data)
#         val_accuracy = eval_metric(val_predict, val_fold_label)
#         fold_accuracies.append(val_accuracy)
#     avg_accuracy = np.mean(fold_accuracies)
#     return avg_accuracy
```

```
[ ]: mnist = np.load(f"data/mnist-data.npz")
     spam = np.load(f"data/spam-data.npz")

     mnist_train_data_flat = mnist["training_data"].reshape(
         mnist["training_data"].shape[0], -1
     )

     mnist_train_d, mnist_train_l, mnist_val_d, mnist_val_l = partition(
         mnist_train_data_flat, mnist["training_labels"], 10000
     )
     spam_train_d, spam_train_l, spam_val_d, spam_val_l = partition(
         spam["training_data"], spam["training_labels"], 0.2
     )
```

```
[ ]: find_best_model(
         "spam", spam_train_d, spam_train_l, spam_val_d, spam_val_l, 10
     )  # Accuracy: 0.8759328855751498, kernel: rbf, C=10, gamma: 0.1
```

```
Accuracy: 0.6990407673860911, kernel: poly, C=10, gamma: 0.001
Accuracy: 0.7278177458033573, kernel: poly, C=10, gamma: 0.01
Accuracy: 0.7649880095923262, kernel: poly, C=10, gamma: 0.1
Accuracy: 0.7685851318944844, kernel: rbf, C=10, gamma: 0.001
Accuracy: 0.8321342925659473, kernel: rbf, C=10, gamma: 0.01
Accuracy: 0.841726618705036, kernel: rbf, C=10, gamma: 0.1
```

```
[ ]: find_best_model(
         "minst", mnist_train_d, mnist_train_l, mnist_val_d, mnist_val_l, 0.000001
     )  # Accuracy: 0.9789, kernel: poly, C=1e-06, gamma: 0.01
```

```
Accuracy: 0.9775, kernel: poly, C=1e-06, gamma: 0.001
Accuracy: 0.9775, kernel: poly, C=1e-06, gamma: 0.01
Accuracy: 0.9775, kernel: poly, C=1e-06, gamma: 0.1
```

0.6.1 Kaggle

```
[ ]: import pandas as pd


     def kaggle_submit(dataset, train_data, train_labels, test_data, kernel, c,␣
      ↪gamma):
         model = svm.SVC(kernel=kernel, C=c, gamma=gamma)
         model.fit(train_data, train_labels)
         if dataset == "mnist":
             test_data = test_data.reshape(test_data.shape[0], -1)

         test_labels = model.predict(test_data)

         test_labels = test_labels.astype(int)
         df = pd.DataFrame({"Category": test_labels})
         df.index += 1
         df.to_csv(f"{dataset}_predictions.csv", index_label="Id")
```

```
[ ]: kaggle_submit("spam", spam_train_d, spam_train_l, spam["test_data"], "rbf", 10,␣
      ↪0.1)
     # kaggle_submit(
     #     "mnist", mnist_train_d, mnist_train_l, mnist["test_data"], "poly", 0.
      ↪000001, 0.01
# )
```